1. **Write a parallel or distributed program that counts the number of subsets of `k` out of `N` that satisfy a given property. You have a function (`bool pred(vector <int> const& v)`) that verifies if a given subset satisfies the property. Your program shall call that function once for each subset of `k` elements and count the number of times it returns true.**

```
#include <iostream>
#include <fstream>
#include <vector>
#include <thread>
#include <atomic>

using namespace std;

atomic <int> cnt;

inline bool check(vector <int> const &v) {
  if(v.size() == 0) {
    return false;
  }
  return v[0] % 2 == 0;
}

inline void generate(vector <int> v, int k, int n, int T) {
  if(v.size() == k) {
    for(auto it : v) {
      cerr << it << ' ';
    }
    cerr << '\n';
    if(check(v)) {
      cnt ++;
    }
    return ;
  }
  int lst = 0;
  if(v.size() > 0) {
    lst = v.back();
  }
  if(T == 1) {
    for(int i = lst + 1; i <= n; ++ i) {
      v.push_back(i);
      generate(v, k, n, T);
      v.pop_back();
    }
  } else {
    thread t([&]() {
      vector <int> newv(v);
      for(int i = lst + 1; i <= n; i += 2) {
        newv.push_back(i);
        generate(newv, k, n, T / 2);
        newv.pop_back();
      }
    });
    vector <int> aux(v);
    for(int i = lst + 2; i <= n; i += 2) {
      aux.push_back(i);
      generate(aux, k, n, T - T / 2);
      aux.pop_back();
    }
```

```
    t.join();
  }
}

int main() {
  generate(vector <int> (), 3, 5, 2);
  cout << cnt << '\n';
}
```

2. **[ALREADY GIVEN] Consider the following code for enqueueing a work item to a thread pool. Find the concurrency issue and fix it. Also, add a mecahnism to end the threads at shutdown.**

```
class ThreadPool {
  condition_variable cv;
  mutex mtx;
  list<function<void()>> work;
  vector <thread> threads;

  void run() {
    while(true) {
      if(work.empty()) {
        unique_lock<mutex> lck(mtx);
        cv.wait(lck);
      } else {
        function<void()> wi = work.front();
        work.pop_front();
        wi();
      }
    }
  }
public:
  explicit ThreadPool(int n) {
    threads.resize(n);
    for(int i = 0; i < n; ++ i) {
      threads.emplace_back([this](){run();});
    }
  }
  void enqueue(function <void()> f) {
    unique_lock<mutex> lck(mtx);
    work.push_back(f);
    cv.notify_one();
  }
}
```

Concureency issue

The issue here is that the ThreadPool is started with `n = 2` and on the main thread, we create another two threads that calles the `enqueue()` method, both will try to access `work.front()` element, and follow with an `work.pop_front()`.

Solution

The critical resource is the list. The correct way would be to have the mutex to also lock the other branch of the is case

Code

```
void run() {
  while(true) {
    if(work.empty()) {
      unique_lock<mutex> lck(mtx);
      cv.wait(lck);
    } else {
      {
        unique_lock<mutex> lck(mtx);
        function<void()> wi = work.front();
        work.pop_front();
      }
      wi();
    }
  }
}
```

3. [ALREADY GIVEN] Write a parallel algorithm that computes the product of two big numbers. Or that computes the scalar product of two vectors (same thing)

```
#include <iostream>
#include <fstream>
#include <vector>
#include <thread>

using namespace std;

inline vector <int> solve(vector <int> a, vector <int> b, int T) {
  vector <int> c;
  int n = a.size();
  int m = 2 * n - 1;
  c.resize(m, 0);
  vector <thread> thr;
  for(int idx = 0; idx < T; ++ idx) {
    thr.push_back(thread([&, idx](){
      // v[i + j] += a[i] * b[j]
      // v[x] = sum(a[i] * b[x - i])
      for(int x = idx; x < m; x += T) {
        for(int i = 0; i < n; ++ i) {
          if(x - i >= n || x - i < 0) {
            continue;
          }
          c[x] += a[i] * b[x - i];
        }
      }
    }));
```

```
  }
  for(int i = 0; i < thr.size(); ++ i) {
    thr[i].join();
  }
  return c;
}

int main() {
  ifstream fin("input.in");
  vector <int> a, b;
  int n;
  fin >> n;
  for(int i = 0; i < n; ++ i) {
    int x;
    fin >> x;
    a.push_back(x);
  }
  for(int i = 0; i < n; ++ i) {
    int x;
    fin >> x;
    b.push_back(x);
  }
  for(auto it: solve(a, b, 5)) {
    cout << it << ' ';
  }
}
```

3. Partial sums

Given a sequence of ’n’ numbers, compute the sums of the first ‘k’ numbers, for each ‘k’ number between ‘1’ and ’n’. Parallelize the computations, to optimise for low latency on a large number of processors. Use at most 2*n additions, but no more than 2*log(n) additions on each computation path from inputs to an output.

Example: if the input sequence is ‘1 5 2 4’ then the output should be ‘1 6 8 12’.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <thread>
#include <assert.h>

using namespace std;

const int maxlg = 22;
const int maxn = 500005;

int n;
long long dp[maxlg][maxn], sum[maxn], psum[maxn];

const int T = 5;

//#define debug
```

```
inline void doIt(int idx) {
    for(int i = idx; i < n; i += T) {
        int act = 0;
        int now = i + 1;
        for(int bit = 0; (1 << bit) <= now; ++ bit)
            if(now & (1 << bit)) {
                sum[i] += dp[bit][act];
                act += (1 << bit);
            }
    }
}

int main() {

    cin >> n;
    int element;
    for(int i = 0; i < n; ++ i) {
        cin >> element;
        dp[0][i] = element;
        psum[i] = psum[i - 1] + dp[0][i];
    }

    for(int k = 1; (1 << k) < maxn; ++ k) {
        for(int i = 0; i < n; ++ i) {
            dp[k][i] = dp[k - 1][i] + dp[k - 1][i + (1 << (k - 1))];
        }
    }

    vector <thread> th;
    for(int i = 0; i < min(T, n); ++ i)
        th.push_back(thread(doIt, i));

    for(int i = 0; i < th.size(); ++ i)
        th[i].join();

    for(int i = 0; i < n; ++ i) {
        cout << sum[i] << '\n';
        assert(sum[i] == psum[i]);
    }

    return 0;
}
```

1. multiplication of two polynoms having both karatsuba and secvential implementations

```
#include <iostream>
#include <fstream>
#include <vector>
#include <ctime>
#include <thread>

using namespace std;

const int T = 3;

int n;
vector <int> a, b, sol;
vector <thread> threads;
```

```
inline void work(int idx) {
    for(int i = idx; i < n; i += T) {
        // solve for position i
        for(int x = 0; x <= i; ++ x) {
            int y = i - x;
            sol[i] += a[x] * b[y];
        }
    }
}

inline void solve() {
    cin >> n;
    for(int i = 0; i < n; ++ i) {
        int x;
        cin >> x;
        a.push_back(x);
    }
    for(int i = 0; i < n; ++ i) {
        int x;
        cin >> x;
        b.push_back(x);
    }
    sol.resize(2 * n - 1, 0);
    for(int i = 0; i < min(n, T); ++ i)
        threads.push_back(thread(work, i));
    for(int i = 0; i < threads.size(); ++ i)
        threads[i].join();
    for (auto x: sol)
        cout << x << " ";
}

int main(int argc, char* argv[]) {
    solve();
}
```

2. **[ALREADY GIVEN] Write a parallel or distributed program that finds all the prime numbers up to N. Hint: serially produce all the prime numbers up to sqrt(N) and distribute them to all threads or processes.**

```
#include <iostream>
#include <fstream>
#include <vector>
#include <thread>

using namespace std;

void solve(int N) {
    vector <thread> thr;

    for (int i = 3; i <= N; i++) {
        thr.push_back(thread([&, i])) {
            bool isPrime = true;
            for (int j = 2; j <= sqrt(i); j++) {
                if i % j == 0 {
                    isPrime = false;
                    break;
                }
            }
            if (isPrime) {
                print(i);
            }
        }
    }

    for(int i = 0; i < thr.size(); ++ i) {
        thr[i].join();
    }
}

int main() {
    ifstream fin("input.in");
    int n;
    fin >> n;
    solve(n);
    return 0;
}
```